Discovering Machine-Specific Code Improvements

Technical Report

S. L. Graham

(415) 642-2059

Contract No. N00039-82-C-0235

November 15, 1981 - December 30, 1985

Arpa Order No. 4031

1461

Discovering Machine-Specific Code Improvements

Peter B. Kessler

1. Abstract

A compiler construction tool has been designed and built that automates much of the case analysis necessary to exploit special purpose instructions on a target machine. Given a suitable description of the target machine, the analysis identifies instruction sequences that are equivalent to single instructions. During code generation, these equivalences can be used to avoid inefficient sequences in favor of more efficient instructions.

A working prototype of the instruction set analyzer needed in the framework outlined by [Giegerich 83] is presented. In contrast to the work presented in [Davidson and Fraser 80, 84], machine descriptions are analyzed entirely during compiler construction (*i.e.*, once per compiler), rather than during code generation (*i.e.*, each time the compiler is used). [RKessler 84] describes such a system for discovering equivalent instructions for instruction sequences of length 2. The techniques presented here can identify instruction sequences of arbitrary length that are equivalent to single instructions.

This analysis has been applied to the descriptions of two machines, and the results have been used to replace hand-written case analysis routines in an otherwise table driven code generator [Henry 84].

2. Motivation

The translation of a programming language onto a target architecture requires analyses of both the language and the architecture. Much of the analysis of programming languages is now formalized and incorporated in compiler construction tools for the "front-ends" of compilers [Lesk and Schmidt 75, Johnson78]. Analysis of a target architecture by construction tools for the "back-ends" of compilers is of at least two kinds. It is sufficient to discover an implementation of each language construct on the target architecture [Henry 84]. Additional analysis may discover features of the architecture that can be exploited to generate more efficient code. This paper describes a method of automating such additional analysis.

Often a target architecture contains general purpose instructions and, in addition, special purpose instructions that perform the same operations for a restricted set of operands (for example, addition versus increment). Such special purpose instructions are often faster or smaller than the equivalent more general instructions. A code generator that avoids less efficient sequences in favor of more efficient equivalent instructions produces better code. The analysis of what restrictions must hold to use special purpose instructions is tedious and prone to error if done by hand, and is susceptible to automation. Such analysis takes a suitable machine description and discovers when sequences of general purpose instructions are equivalent to special purpose instructions. One may think of the analysis as imposing a set of constraints on general purpose instructions that make them equivalent to a special purpose instruction.

3. Scope

This paper describes a novel technique for identifying three kinds of *idioms*: set idioms, binding idioms, and composite idioms. (I use the term idiom as an analogy to idioms in natural languages: phrases, particular to the language, that are used in place of more general terms.)

A ***set idiom*** is a special purpose instruction that can be used to replace a general purpose instruction when one (or more) operand of the general purpose instruction has a value from a particular (machine-specific) set. For example, an increment-by-1 instruction may be a set idiom for a general addition instruction when one of the addends of the addition has the value 1. The increment example restricts the operand to a particular value though many architectures have instructions to perform operations with an operand from a small set of values.

Binding idioms refer to special purpose instructions that perform the same operation as more general purpose instructions when two (or more) operands of the general instruction reference the same storage location. By choosing a suitably general operand addressing notation, one can model a large class of popular operand addressing schemes.

A ***composite idiom*** is an instruction that performs the same computations as a sequence of instructions. For example, many architectures have instructions for use at the ends of loops. Such an instruction might add a step value to an index, compare the index against a limit value, and branch on the result of the comparison. Special purpose loop instructions are often preferable to separate addition, comparison, and branch instructions.

Often these three types of semantic restrictions must be considered together to discover equivalent code sequences in a target architecture. For instance, in the previous example, the loop control instruction might only increment the index value by 1, as opposed to allowing unrestricted addition. In addition, the same operand must be tested as was incremented for the sequences to be equivalent.

4. Decomposition of Instruction Descriptions

Idioms are identified by imposing constraints on instruction sequences so that they perform the same computation as a single instruction from the target machine. That is, given any single instruction, I identify all the other (reasonable) sequences of code that can be replaced by that instruction. This process is repeated for each instruction on the target machine, yielding a list of all the constrained equivalences. The assumption here is that the single instructions of the target machine are implemented efficiently. This assumption is verified by comparing the costs of the alternatives.

The identification of idioms is driven by the general classes of idioms given above and the particulars of instructions on the target machine. If there is an instruction on the target machine that adds small constants, a search will be made for more general addition instructions that can be restricted to add small constants. The search for idioms is not driven by rules like "look for instructions that add small constants", or "look for instruction sequences for the ends of loops".

4.1. An Example

As an example, consider the problem of generating VAX-11[1] code for the source fragment:

```
foo := foo + 1;
if foo = 0 then ...
```

That is, an increment of a variable by 1, followed by a comparison of that variable to 0. A naive code generator might generate separate, unrestricted, instructions for addition and comparison:

```
addl3  $1,foo,foo     -- foo ← foo + 1
cmpl   foo,$0         -- cc-z ← foo = 0
```

The sequence **addl3** (three operand long integer addition) followed by **cmpl** (two operand long integer comparison) with operands constrained as in the example above is equivalent to the VAX-11 instruction:

```
incl   foo            -- foo ← foo + 1; cc-z ← foo = 0
```

(one operand long integer increment by 1). The latter sequence is both shorter and faster than the former, and so should be preferred. The constraints that produce this equivalence can all be discovered by examining a suitable description of the VAX-11 architecture.

The derivation of constraints on equivalence is done during compiler construction. All properties of the target machine that affect equivalence are considered during this analysis, leaving only properties of the program to be checked at compile time. For example, the fact that **addl3** and

[1] VAX-11 is a trademark of Digital Equipment Corporation.

incl both do long integer addition is discovered at compiler construction time, as is the required reuse of the operand as shown in the example above. Whether the operands in a particular program obey those constraints cannot be checked until compile time.

4.2. Machine Descriptions

The effect of the VAX-11's one operand long integer increment-by-1 instruction, **incl**, can be described by the sequence of computations:

$$\begin{aligned}
&\text{i-dest} \leftarrow \text{i-dest} + 1;\\
&\text{cc-n} \leftarrow \text{i-dest} < 0;\\
&\text{cc-z} \leftarrow \text{i-dest} = 0;\\
&\text{cc-v} \leftarrow \text{overflow}(\text{i-dest} + 1);\\
&\text{cc-c} \leftarrow \text{carry}(\text{i-dest} + 1);
\end{aligned}$$

where 'i-dest' is a reference to the operand of the instruction, and the various 'cc's are the condition codes that the VAX-11 sets during most arithmetic instructions. (Many details of the instructions have been suppressed in these descriptions in an attempt at clarity.) Similarly, the three operand unrestricted long integer addition instruction, **addl3**, can be described by:

$$\begin{aligned}
&\text{a-dest} \leftarrow \text{a-src}_2 + \text{a-src}_1;\\
&\text{cc-n} \leftarrow \text{a-dest} < 0;\\
&\text{cc-z} \leftarrow \text{a-dest} = 0;\\
&\text{cc-v} \leftarrow \text{overflow}(\text{a-src}_2 + \text{a-src}_1);\\
&\text{cc-c} \leftarrow \text{carry}(\text{a-src}_2 + \text{a-src}_1);
\end{aligned}$$

and the two operand long integer compare instruction, **cmpl**, can be described by:

$$\begin{aligned}
&\text{cc-n} \leftarrow \text{c-src}_1 < \text{c-src}_2;\\
&\text{cc-z} \leftarrow \text{c-src}_1 = \text{c-src}_2;\\
&\text{cc-v} \leftarrow \text{FALSE};\\
&\text{cc-c} \leftarrow \text{FALSE};
\end{aligned}$$

My analysis does not rely on semantics associated with most operators in the machine descriptions; the majority of the analysis considers only the syntax of the descriptions. The compiler writer has the task of using the same notation for the same operations, and distinguishing operations by using distinct symbols for each. The only semantics required for the examples in this paper are the sequencing of computations (via the ';' operator, which is used in this paper in place of the order-dependent and order-independent separators available in the real description language), assignment (represented by the '←' operator), and references to operands and constants (which in reality require some syntax, but in these examples should be apparent from the context).

Note that simple syntactic comparison of instruction descriptions is not sufficient, since architectures rarely have multiple instructions that perform exactly the same computations. Rather, the syntactic mismatches are used to identify semantic restrictions that must be imposed to achieve equivalence. Where a syntactic mismatch identifies a property of the program that can be constrained to achieve equivalence, that constraint is recorded during compiler construction and is checked at compile time.

4.3. An Example Decomposition

Consider the decomposition of an **incl** instruction into the sequence **addl3** followed by **cmpl**. Decomposition proceeds from the end of an instruction description towards the beginning. Each instruction on the target machine is examined to see if it can be constrained to match the tail of the **incl**. Among those instructions is the **cmpl** instruction, which can be constrained to match the condition code settings of the **incl** instruction. The remaining unmatched portion of the **incl** instruction is the addition of 1. The decomposition algorithm again searches the target machine description for instructions that perform additions. Among those instructions is the **addl3** instruction, with appropriate constraints on its operands. In this way the **incl** instruction is "decomposed" into (among other sequences) an **addl3** followed by a **cmpl**.

The preceding description glosses over many details in the hope of conveying the intuition behind decomposition. Most of the details have to do with the imposition of constraints, and are explained below.

In the comparison of the condition code settings of the **incl** instruction against those in the **cmpl** instruction, several mismatches occur. For example, the computation

$$\text{cc-n} \leftarrow \text{i-dest} < 0;$$

from the **incl** syntactically matches the computation

$$\text{cc-n} \leftarrow \text{c-src}_1 < \text{c-src}_2;$$

from the **cmpl**, except for the references to the operands. The mismatch of 'i-dest' against 'c-src_1' records the constraint that the destination of the **incl** must be the same as the first source operand of the **cmpl**. This is an example of an operand binding constraint. The mismatch of '0' against 'c-src_2' records the constraint that the second source of the **cmpl** must be the constant 0. These constraints are rediscovered (but not further constrained) by the examination of the setting of 'cc-z'.

The settings of 'cc-v' and 'cc-c' cannot be made equivalent by constraining operands. However, if 'cc-v' and "cc-c' could be shown to be dead during code generation, then the discrepancy in the instruction descriptions would be irrelevant and could be ignored during compiler construction. Thus the matching of the assignments to 'cc-v' and 'cc-c' records the constraint that these values must be dead for the equivalence to be valid.

All that remains to be covered of the **incl** instruction is the long word addition. The examination of the target machine instructions discovers that **addl3** performs long word addition and attempts to extend the decomposition by comparing the description of **addl3** with the remainder of the **incl**. The assignments to the condition codes by the following **cmpl** instruction will cause the condition codes to be dead at the prepended **addl3** instruction, and so the condition code settings from the **addl3** description may be pruned before the **addl3** is matched to the remainder of the **incl**. The match of

$$\text{i-dest} \leftarrow \text{i-dest} + 1;$$

from the **incl**, against

$$\text{a-dest} \leftarrow \text{a-src}_2 + \text{a-src}_1;$$

from the **addl3**, derives the constraints that (1) the destinations of the **incl** and **addl3** must be the same, (2) that the destination of the **incl** must be the same as the second source operand of the **addl3**, and (3) that the first source operand of the **addl3** must be the constant 1. These constraints, together with the ones derived above, are sufficient to ensure the equivalence of the two instruction sequences.

5. Discussion

Instruction sequences may be extended to arbitrary lengths in the attempt to decompose an instruction. This is a major contribution of the decomposition technique. The complexity of the analysis process is exponential in the number of instructions on the target machine, with the degree of the exponential depending on the lengths of the sequences found to be equivalent. The sequences do not grow very long, since most architectures do not include extremely complex instructions (that can be decomposed by this algorithm). A performance improvement is achieved by matching the "tails" of sequences only once. For example, once **cmpl** has been determined to match the tail of **incl**, any prepended instructions, like **addl3**, only examine the unmatched remainder of the **incl**. In addition, trial extensions that fail (due to mismatches of the architecture or unsatisfiable constraints on the programs) are not extended further. Thus, the number of sequences considered in practice is consi

The algorithm demonstrated above works from the end of the instruction towards the beginning. A more straightforward technique is to proceed forward through the instruction descriptions, using code generation techniques to discover alternative implementations. Forward decomposition is too "greedy" to find certain decompositions, however. Consider the forward decomposition of **incl**. The first round of the algorithm would match **incl** with (among other instructions) **addl3**. However, **addl3** can be constrained to completely match **incl**, as an examination of the descriptions will reveal. Why should forward decomposition ever consider appending an **cmpl** instruction when the **incl** can be completely matched by **addl3**? (Note that no classification of "redundant test" instructions is possible *a priori*).

The descriptions of the machine include the descriptions of the computations performed by addressing modes. Thus decomposition may discover that computations implicit in operand addressing may be used to replace explicit instructions. For example a move-effective-address instruction with a source operand that adds a constant displacement to a register can be used as an alternate implementation for the addition of a constant, provided the other addend is a register.

The decomposition algorithm discovers code sequences that perform equivalent computations. The sequences are often not equivalent in cost (the difference in costs is the motivation for identifying otherwise equivalent sequences!). Both code space and execution time must be considered. Accurate costs for sequences cannot be compared during analysis at compiler construction time, however. In part this is because instructions are analyzed for correspondence of operands, without necessarily restricting operands to particular addressing modes. Therefore, the costs for operands can not be accurately determined until a particular program is compiled. As an extreme case, the VAX-11 has equivalent code sequences where the choice of which sequence is best depends on the compile time value of an operand displacement. Thus, cost functions can not be analyzed during compiler construction.

6. Related Work

Recent grammar-based code generator generation techniques rely on hand-written case analysis to select instructions with semantic restrictions. For example, [Henry 84] requires hand-written routines to use these instructions. [Ganapathi 80] uses attribute-influenced parsing to provide a framework for hand-written attribute functions that select instructions with semantic restrictions. The work presented here automates the analysis of the target machine for opportunities to exploit such instructions. The results of such an analysis can be used after code generation to improve generated code, or can be incorporated as attribute tests during code selection.

The techniques proposed in this paper fit into the framework outlined in [Giegerich 83]. Giegerich proposes that machine descriptions be examined during compiler construction for instances of "standard rules" of code generation. Rather than using a list of such axioms, my technique uses the instructions of the target machine as starting points for discovering equivalent instruction sequences.

Much of Giegerich's analysis discovers the effects of instructions on the data flow of programs, and in particular, occasions when the data flow is invariant (or at least still "safe") in spite of transformations of instructions. I borrow some of Giegerich's data flow results in my algorithms.

Previous examinations of machine descriptions for special purpose instructions have composed instruction sequences for analysis. Using composition, sequences of presumed inefficient code are constructed, and then the machine description is consulted to find a more efficient implementation of that code. Davidson and Fraser use sequences that occur during compilation, and perform the analysis of the target machine at that time [Davidson and Fraser 80]. Robert Kessler describes a system that considers sequences composed during compiler construction [RKessler 84]. Analysis of the target machine description during compiler construction allows a more extensive search for equivalent instructions, with, however, less precise information about particular programs. Davidson and Fraser have recently taken to caching the results of their analysis, thus they can avoid rediscovering equivalences during a single compilation, as well as between several compilations [Davidson and Fraser 84].

Using the composition algorithms, sequences must be composed before the machine description is examined. Thus, the number of sequences examined is an exponential function of the number of target machine instructions, whose degree is the length of the sequences considered. The composition algorithms are limited in practice to considering pairs of instructions. The composition analysis thus finds only 1-to-1 and 2-to-1 equivalences.

7. Implementation

I have constructed a tool to analyze machine descriptions by decomposition. The current organization generates a table of instruction sequences, their equivalents, and the constraints on the equivalence. This table is used to retarget an improver of assembler code for a retargetable compiler. The same information could be used to affect the selection of efficient code in the code generator itself, rather than transforming the output of the code generator. Our standard retargetable code generator normally makes use of hand-written code improvement routines [Henry 84]. The table driven improver replaces those routines, making the compiler more easily retargeted to new architectures.

I have used this tool to analyze two architectures, the VAX-11 and the M68000. The analysis of the VAX-11 takes just over 2 VAX-11/750 cpu hours and discovers almost 1300 idioms. The analysis of the M68000 takes just under 4 VAX-11/750 cpu hours and discovers over 500 idioms. The longest sequences discovered were of length 3 on both architectures.

The analyzer exploits several properties of the machine descriptions to reduce the amount of work required of it. For example, families of instructions that vary only in the type of their operands can often be analyzed only once. In addition, many instructions in a target architecture can be shown to perform unique operations, and thus there is no need to decompose them or to use them in the decomposition of other instructions.

8. Conclusion

The addition of retargetable code improvers to the suite of compiler construction tools improves the overall quality of the compilers. The uniform application of such tools provides a standard of code generator quality, which makes it possible to compare machine architectures. The availability of compilers that can exploit special purpose instructions frees machine architects to design such instructions into new machines.

This paper describes a novel technique for analyzing machine descriptions for opportunities to use an interesting class of special purpose instructions. This work is an addition to the growing collection of compiler construction tools. The use of such tools simplifies the retargeting of a compiler.

9. References

[Davidson and Fraser 80]
J. W. Davidson and C. W. Fraser, "The Design and Application of a Retargetable Peephole Optimizer", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2, pp. 191-202, April 1980.

[Davidson and Fraser 84]
J. W. Davidson and C. W. Fraser, "Automatic Generation of Peephole Optimizations", Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, Montreal, Quebec, SIGPLAN Notices, Montreal, Quebec, Vol. 19, No. 6, pp. 111-116, June 1984.

[Ganapathi 80]
M. Ganapathi, "Retargetable Code Generation and Optimization Using Attribute Grammars", PhD. Dissertation, Technical Report #406, Computer Science Department, University of Wisconsin, Madison, 1980.

[Giegerich 83]
R. Giegerich, "A Formal Framework for the Derivation of Machine-Specific Optimizers", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, pp. 478-498, July 1983.

[Henry 84]
Robert R. Henry, "Graham-Glanville Code Generators", PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, Report No. UCB/CSD 84/184, May, 1984.

[Johnson 78]
S. C. Johnson, "YACC: Yet Another Compiler-Compiler", Bell Laboratories Murray Hill, NJ, July 1978.

[RKessler 84]
Robert R. Kessler, "Peep — An Architectural Description Driven Peephole Optimizer", Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, Montreal, Quebec, SIGPLAN Notices, Montreal, Quebec, Vol. 19, No. 6, pp. 106-110, June 1984.

[Lesk and Schmidt 75]
M. E. Lesk and E. Schmidt, "LEX — A Lexical Analyzer Generator", Computer Science Technical Report TR-39, Bell Laboratories Murray Hill, NJ, October, 1975.